EXHIBIT 11

Confuzzled

- Home
- Learning
  - Tutorials
    - Data Structures
      - Andersson Trees
      - AVL Trees
      - Binary Search Trees I
      - Binary Search Trees II
      - Hash Tables
      - Red Black Trees
      - Skip Lists
      - Linked Lists
    - Algorithms
    - Languages
  - Articles
    - Asymptotic Notation
    - Using rand()
  - Libraries
- Licensing

Welcome back! Or if this is your first experience with my tutorials, get ready for a good time. But first, why another red black tree tutorial? Anyone who searches for red black trees on Google will be rewarded with a slew of resources that include tutorials, general descriptions, Java applets, papers, libraries, and power point presentations. This is good, but not good enough. Most resources are content with only looking at insertion (slackers!). Others fail to describe deletion in such a way that one can easily understand why it works, obviously thinking that enumerating the cases should be enough. Even worse, every resource fails to notice that red black trees can be written several different ways.

Well guys, I'm not that smart. Insertion into a red black tree is painfully simple. Any programmer with half a brain and a little experience with binary search trees can figure it out with minimal effort. Deletion on the other hand is, quite frankly, a pain in the ass. I had better luck independently rediscovering the cases than trying to figure out why some of the traditional cases are used. For example, a sibling should be black, so a special case makes it so, yet nobody seems to know why. After dealing with that case, I can both tell you and show you why (it's not called a case reduction for nothing).

When it comes to red black trees, there is only one widely accepted resource: "Introduction to Algorithms" by Cormen, Leiserson, and Rivest, affectionately known as CLR. This is a good resource because not only does it describe both insertion and deletion, it gives sufficient pseudocode for both operations! As the only well known and easily accessible text that does this, just about every red black tree in the real world uses a translation of the CLR algorithms. Unfortunately, CLR's approach is both complicated and inefficient, through heavy use of parent pointers. Red black trees don't have to be that hard.

I learned red black trees the hard way. Given only a description of the rules, I worked through every case I could think of while trying to devise an algorithm. Without a copy of CLR (I still don't own it), I wasn't able to see how the problem was meant to be solved. This is a good thing because I "discovered" several ways of doing it, and I believe that I peeled away most of the complexity. I cut myself on the sharp edges of the algorithms, I wept from frustration trying to get them to work, and now I'll share the results of that effort so that you don't have to go through the same ordeal.

This tutorial will cover both the non-recursive top-down and recursive bottom-up algorithms for red black insertion and deletion, as well as provide full C source code for all variations described. The most common implementation is based off of the pseudocode from CLR, but I won't cover that because it's just another implementation of the bottom-up process. If you're interested in how to implement red black trees with parent pointers, either CLR or Ben Pfaff's excellent online book will serve you well. If you're interested in a non-recursive red black implementation using an explicit stack instead of recursion, Ben Pfaff's online book also covers this. I see no point in reproducing something that is dealt with adequately elsewhere.

http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx Go SEP NOV JAN
125 captures 29
12 Feb 2007 - 8 Feb 2021 2013 2014 2015
About this capture

Basic binary search trees are simple data structures that boast O(log N) search, insertion, and deletion. For the most part this is true, assuming that the data arrives in random order, but basic binary search trees have three very nasty degenerate cases where the structure stops being logarithmic and becomes a glorified linked list. The two most common of these degenerate cases is ascending or descending sorted order (the third is outside-in alternating order). Because binary search trees store their data in such a way that can be considered sorted, if the data arrives already sorted, this causes problems. Consider adding the values 0,1,2,3,4 to a binary search tree. Since each new item is greater in value than the last, it will be linked to the right subtree of every item before it:

```
0
 \
  1
   \
```

```
        2
         \
          3
           \
            4
```

That's not a very good binary search tree. The performance degrades from O(log N) to O(N) because the tree is now effectively a linear data structure. We would rather have two choices at each node instead of just one. This way we can take full advantage of the two-way design of binary search trees:

```
   1
 /   \
0     3
    /   \
   2     4
```

Many brain cells have perished trying to come up with an easy way to guarantee this optimum structure, which is called a balanced binary search tree. Unfortunately, it takes far too much work to maintain a perfectly balanced binary search tree, so it's impossible to efficiently guarantee perfect balance at all times. However, we can come close enough to guarantee logarithmic performance. Several of these "close enough" balancing schemes are in common use, and the two forerunners are AVL trees and red black trees. Because I have another tutorial on AVL trees, we will only cover red black trees in detail here.

Red black trees were originally an abstraction of an abstraction that managed to become a concrete data structure of their own. The original abstraction was suggested by Rudolf Bayer, which he called symmetric binary B-trees. The idea was to simulate a B-tree of

...ies, all leaves ... not a binary

The basic idea behind the symmetric binary B-tree is that a node can have horizontal or vertical links. In this way a binary search tree can simulate the structure of B-tree nodes. A vertical link separates two different nodes and a horizontal link separates nodes that are treated logically as the same B-tree node. The equivalent B-tree and symmetric binary B-tree (SBB-tree) nodes are as follows (* = Unknown node). Notice how the B-tree node's structure is faked using multiple binary tree nodes:

```
 2-node

 SBB-tree      B-tree

    0            0
  /   \   ->   /   \
 *     *      *     *

 3-node

 SBB-tree 1         SBB-tree 2           B-tree

       1                0                 0   1
     /   \            /   \             /   |   \
    0     *   or    *       1     ->  *     *     *
  /   \                   /   \
 *     *                 *     *

 4-node

     SBB-tree 1            B-tree

         1               0     1     2
       /    \          /    |     |    \
```

```
 0       2     => *    *    *    *
 / \    / \
*    * *    *
```

The algorithms for maintaining B-trees are confusing at best, probably because they grow up instead of down and programmers have trouble with that. As a result, the algorithms for B-trees don't translate well into the symmetric binary B-tree abstraction and are still confusing. So while it was a good idea, symmetric binary B-trees were missing something.

Later, Robert Sedgewick and Leonidas Guibas came up with a mnemonic abstraction that made symmetric binary B-trees easier to understand (and the name was shorter too!). By giving nodes a color, you could easily differentiate between vertical and horizontal links. Under this abstraction, nodes that are part of a logical B-tree node (horizontal links) are colored red, while nodes that separate logical B-tree nodes (vertical links) are colored black. Thus was born the red black tree. Here are the 2, 3, and 4-nodes


```
  0,B
 /   \
*     *


 3-node

        1,B            0,B
       /   \          /   \
    0,R     *   or   *     1,R
   /   \                  /   \
  *     *                *     *


 4-node

         1,B
       /     \
    0,R       2,R
   /   \     /   \
  *     *   *     *
```

This gives us some easy to remember rules to maintain balance. First, a node that's colored red can't have a child that's also colored red, because that doesn't fit into the abstraction. If a node is colored red, then it's part of a logical node, and the only possible color for the next node is black because a red node's links play the part of a B-tree node's links to the next B-tree node. A violation of this rule is called a red violation (duh!).

Next, because in a B-tree any path from the root to a leaf always has the same number of nodes, and all leaves are on the same level, the number of black nodes along any path in a red black tree must be the same. Because a black parent and its red children are a part of the same logical node, red nodes aren't counted along the path. This is called the black height of a red black tree. A violation of this rule is called a black violation.

Finally, the root of a red black tree must be black. This makes all kinds of sense because only a black node can have red children, and the root has no parent. However, this rule only applies when using the B-tree abstraction. In practice the root can be either red or black and no affect the performance of a red black tree.

All algorithms that maintain balance in a red black tree must not violate any of these rules. When all of the rules apply, the tree is a valid red black tree, and the height cannot be any shorter than log(N + 1) but also no taller than 2 * log(N + 1). This is obviously logarithmic, so a red black tree guarantees approximately the best case for a binary search tree.

Now, because red black trees are so common these days, the original abstraction has been largely forgotten in favor of a data structure that simply avoids violating two basic

... ne number of black nodes. This fulfills the requirements of a red black tree ... maintain the ... me, but I' ... does make the algorithms simpler because you can avoid the special case of a red violation at the root.

Before we start, we'll need to cover some of the basics and get a little framework code going. First, the node and tree structures. However you want to represent a node's color is up to you. A common solution is an enumeration with with two constants, one for black and one for red. I find it easier to use a flag where if the flag is set, the node is red and if the flag is not set, the node is black. Naturally the inverse works as well, but the flag solution with red being set will be what this tutorial uses:

```
struct jsw_node {
  int red;
  int data;
  struct jsw_node *link[2];
};

struct jsw_tree {
  struct jsw_node *root;
};
```

I'm sure that if this is your first time reading one of my tree tutorials that the link array could be confusing. Instead of using two pointers called left and right, as is the usual convention, I prefer to use an array where index 0 is left and index 1 is right. This makes it easier to merge the symmetric cases together to shorten the code and to guarantee correctness. This is my personal idiom, and to the best of my knowledge, I'm the only one who merges symmetric cases using it. To minimize shock, here is the basic difference between the "classic" idiom and my own:

```
/* Classic binary search tree insertion */
struct cbt_node *cbt_insert ( struct cbt_node *root, int data )
{
  if ( root == NULL )
    root = make_node ( data );
  else if ( data < root->data )
    root->left = cbt_insert ( root->left, data );
  else
    root->right = cbt_insert ( root->right, data );

  return root;
}

/* Julienne Walker's binary search tree insertion */
struct jsw_node *jsw_insert ( struct jsw_node *root, int data )
{
  if ( root == NULL )
    root = make_node ( data );
  else {
    int dir = root->data < data;
    root->link[dir] = jsw_insert ( root->link[dir], data );
  }

  return root;
}
```

At this level there appears to be no savings, but if you add 50 lines of rebalancing code to both the left and right direction cases, it's easy to see that the classic insertion requires 100 extra lines, but my insertion only needs 50 because the two symmetric cases were merged into one. In the actual implementation this results in dir being the direction that we're going and !dir being the opposite direction. Once you get a feel for the idiom, it's not terribly hard to read code that uses it.

Back to the framework: to avoid constantly cluttering up the code ... dary tests, a ... impler. We will assume that a leaf is black, because our leaves are null pointers and trying to fix a red violation with a null pointer scares the hell out of me. :-)

```
int is_red ( struct jsw_node *root )
{
  return root != NULL && root->red == 1;
}
```

Finally, the rotations. Any tree balancing scheme will use rotations to change the structure without breaking any of the rules of binary search trees. However, in this case it's cleaner to also handle recoloring of the nodes that are rotated. A single rotation in a red black tree simply rotates the nodes as normal, then sets the old root to be red and

the new root to be black. A double rotation is just two single rotations. This is sufficient for insertion, but deletion is harder (what a shock). Despite that, we can still benefit from the color change during rotations in the deletion algorithm, so we will place it in the code for a single rotation, and the double rotation will use it implicitly:

```
struct jsw_node *jsw_single ( struct jsw_node *root, int dir )
{
  struct jsw_node *save = root->link[!dir];

  root->link[!dir] = save->link[dir];
  save->link[dir] = root;

  root->red = 1;
  save->red = 0;

  return save;
}

struct jsw_node *jsw_double ( struct jsw_node *root, int dir )
{
  root->link[!dir] = jsw_single ( root->link[!dir], !dir );
  return jsw_single ( root, dir );
}
```

Don't get so anxious, we're not going to look at insertion just yet. Balanced trees are not trivial data structures, and red black trees are exceptionally tricky to get right because a small change in one part of the tree could cause a violation in a distant part of the tree. So it makes sense to have a little tester function to make sure that no violations have occurred. Why? Because it might seem like the algorithms work for small trees, but then they break on large trees and you don't know why. With a tester function, we can be confident that the algorithm works, provided we slam it with enough data into a big enough tree (too large of a case to test by hand). Since the tester would only be used for debugging, we can use recursion and expect it to be slow, which it is:

```
int jsw_rb_assert ( struct jsw_node *root )
{
  int lh, rh;

  if ( root == NULL )
    return 1;
  else {
    struct jsw_node *ln = root->link[0];
    struct jsw_node *rn = root->link[1];

    /* Consecutive red links */
    if ( is_red ( root ) ) {
      if ( is_red ( ln ) || is_red ( rn ) ) {
        puts ( "Red violation" );
```



```
    rh = jsw_rb_assert ( rn );

    /* Invalid binary search tree */
    if ( ( ln != NULL && ln->data >= root->data )
      || ( rn != NULL && rn->data <= root->data ) )
    {
      puts ( "Binary tree violation" );
      return 0;
    }

    /* Black height mismatch */
    if ( lh != 0 && rh != 0 && lh != rh ) {
      puts ( "Black violation" );
      return 0;
    }

    /* Only count black links */
    if ( lh != 0 && rh != 0 )
      return is_red ( root ) ? lh : lh + 1;
    else
      return 0;
  }
}
```

This algorithm is relatively simple. It walks over every node in the tree and performs certain tests on the node and its children. The first test is to see if a red node has red children. The second test makes sure that the tree is a valid binary search tree. The last test counts the black nodes along a path and ensures that all paths have the same black height. If **jsw_rb_assert** returns 0, the tree is an invalid red black tree. Otherwise it will return the black height of the entire tree. For convenience, a message will also print out telling you which violations occurred. :-)

Now we're ready to move on to insertion! So roll up your sleeves and get ready to mess about in the muck.

Bottom-up Insertion

When we want to insert into a red black tree, we immediately have a decision. Do we color a new node red or black? What kind of algorithm we write depends on this decision because of the violations that it could introduce. If the new node is black then inserting it into the tree always introduces a black violation. The rest of the algorithm would then need to concentrate on fixing the black violation without introducing a red violation. On the other hand, if the new node is red, there is a chance that it could introduce a red violation. The rest of the algorithm would then need to work toward fixing the red violation without introducing a black violation.

But wait! If the new node is red, and it's inserted as the child of a black node then no violations occur at all, whereas if the new node is black, a black violation always occurs. So the logical choice is to color the new node red because there is a possibility that insertion won't violate the rules at all. The same can't be said of inserting a black node. Red violations are also more localized and thus, easier to fix. Therefore, we will give new nodes the color red, and fix red violations during insertion. Laziness wins! How about a helper function that returns a new red node?


```
    rn->data = data;
    rn->red = 1; /* 1 is red, 0 is black */
    rn->link[0] = NULL;
    rn->link[1] = NULL;
  }

  return rn;
}
```

Ah, finally. Insertion. Adding a node. Instead of just talking big about red black trees, we can play with the real thing now. Okay, let's insert a node into a basic binary search tree, that's the first order of business. Then we can walk back up and rebalance if there's a red violation. For simplicity (hah!) we'll use recursion to insert a new node and deny duplicates. Don't forget to notice the last bit of code to make the root black. That's important because it ensures the last red black tree rule and saves us from dealing with a red violation at the root. The code should be obvious, and if it isn't, you're in the wrong tutorial. Try Binary Search Trees I, third door to the left:

```
struct jsw_node *jsw_insert_r ( struct jsw_node *root, int data )
{
  if ( root == NULL )
    root = make_node ( data );
  else if ( data != root->data ) {
    int dir = root->data < data;

    root->link[dir] = jsw_insert_r ( root->link[dir], data );

    /* Hey, let's rebalance here! */
  }

  return root;
}

int jsw_insert ( struct jsw_tree *tree, int data )
{
  tree->root = jsw_insert_r ( tree->root, data );
  tree->root->red = 0;
  return 1;
}
```

This is about as simple as it gets for a binary search tree, but since we know that we might need to rebalance back up the tree, naturally the code to rebalance would go after the recursive call. Remember those programming classes where you had to print numbers in reverse using recursion? The same principle applies here. We recurse down, down, down (down! sit! stay!), then insert a new node at the first leaf we get to (because the second leaf we get to would probably seg fault). Then the recursion rewinds on its own and we can take advantage of that.

So how do we test for a red violation? Simple, if any child on the path is red, test its children to see if one or the other is red. If so, fix the violation. Note that only one child will be red, otherwise something else is broken with the tree because a red violation would had to have already been present and the tree would already have been an invalid red black tree. But we'll assume that it's valid, because that's easier.

Okay, okay, but how do we FIX a red violation? That's equally simple! There are only three cases, and all of them are as easy as falling down. In the first case, we check both children of the node (remember that we don't do this for the new node). If both children are red then the parent has to be black, so we can just flip the colors. The parent


```
          /   \          /   \

        1,R     3,R  ->  1,B     3,B

      /               /

    0,R             0,R
```

Okay, that takes care of the immediate problem, but if the parent's color changes then we risk a violation further up the tree. This is important, if the children of a node don't change color and the parent of a node doesn't change color, there's no way a red violation could propagate. So if 2 remained black in the above diagram, we could terminate. But since it changed from black to red, it's possible that its parent could also be red and there's another violation. So after this case we move up and look suspiciously around for problems.

The second case is if the node's (1 in this case) sibling (3 in this case) is not red. If so, the color flip trick won't work because that would increase the black height of the paths down to the right of 2. So we need to do more work, and that screams rotation. In the second case, 1's left child caused the violation, so we single rotate around 2 to the right, make 2 red, make 1 black, and call it a day:

```
            2,B              1,B

          /   \            /   \

        1,R     3,B  ->  0,R     2,R

      /                            \

    0,R                              3,B
```

This fixes the red violation, the subtree's root doesn't change color, so we can be sure that the red violation won't propagate upward. But it gets better! This rotation doesn't change the black height of either subtree, so the tree is now balanced and we're done with all rotations! But why does the black height not change? Notice that in the "before" diagram, the black height of 2's left subtree is 1 (including 2 itself) and the black height of 2's right subtree is 2. Now look at the "after" diagram. The left subtree's black height is still 1 and the right subtree's black height is still 2! Don't forget to ignore red nodes in the black height, they don't count.

Yes, good catch, that tree is not a valid red black tree. Why? Because the black heights are not the same. What we have here is a black violation! Durnit! But the good news is that this will never happen. The example was contrived to show you how the rotation works without having to draw a large tree, but that exact tree will never occur in the wild. Why? Because there was a black violation before 0 was inserted! In fact, the first rotation in a red black tree of ascending numbers would only be after the seventh number. Let's look at that case after 7 is inserted:

```
        1,B

      /   \

    0,B     3,R

          /   \
```

http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx Go SEP OCT JAN 29 2013 2021 2015
125 captures
12 Feb 2007 - 8 Feb 2021
About this capture

```
                    \
```

7,R

At this point a color flip will be made because 5, 4, and 6 meet the first case. A color flip could cause a red violation higher up, so we move up to 3 and see that it actually did cause a red violation at 3, and now the single rotation case applies. Let's look at the tree before the rotation is made:

```
      1,B
    /     \
 0,B       3,R
         /     \
      2,B       5,R
              /     \
           4,B       6,B
                        \
                         7,R
```

The most important thing to notice right now is that there is NO black violation! Every path has two black nodes, so the only violation is the red violation at 3. When we rotate at 1, 3 becomes the new black root, and 1 becomes red. The resulting tree is as follows. Notice that there are no red violations, and all black heights are the same. So the single rotation case works:

```
            3,B
         /        \
      1,R          5,R
    /     \      /     \
 0,B       2,B  4,B     6,B
                           \
                            7,R
```

It's pretty easy to see now why **jsw_single** sets colors the way it does. The final case is if instead of the left child being red, the right child is red. Well, if the right child is red then a single rotation won't cut it. This is where a double rotation is needed:

```
     2,B                    2,B                  1,B
   /     \                /     \              /     \
 0,R      3,B  ->      1,B       3,B  ->   0,R        2,R
   \                  /                                  \
    1,R            0,R                                    3,B
```


could introduce a black violation, so we continue the double rotation to avoid that case. See if you can find a valid red black tree where this case would exist and make sure that it actually works like we did above. In the meantime, we'll be here moving along without you. But don't worry, it's not like the tutorial will start to disappear if you stop along the way to work the exercises. :-)

The three cases have been looked at, now let's fill in that comment with the code to test for and fix all three cases. Remember that the new node is never rebalanced; the algorithm just returns the new node when it reaches a leaf and leaves rebalancing up to the nodes above. We make sure that if there's a violation, it's in the subtree. That way we can pull of the rotations without setting a status flag that reminds the algorithm to rotate further up. In fact, no status flag is needed at all! It's possible that we could do color flips all the way back up the tree even though only one single or double rotation might be made:

```
struct jsw_node *jsw_insert_r ( struct jsw_node *root, int data )
{
  if ( root == NULL )
```

Title: Eternally Confuzzled - Red Black Tree Tutorial
Link: https://web.archive.org/web/20141129024312/http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx
Captured: Friday, March 12, 2021 15:53:10

```
    root = make_node ( data );
  else if ( data != root->data ) {
    int dir = root->data < data;

    root->link[dir] = jsw_insert_r ( root->link[dir], data );

    if ( is_red ( root->link[dir] ) ) {
      if ( is_red ( root->link[!dir] ) ) {
        /* Case 1 */
        root->red = 1;
        root->link[0]->red = 0;
        root->link[1]->red = 0;
      }
      else {
        /* Cases 2 & 3 */
        if ( is_red ( root->link[dir]->link[dir] ) )
          root = jsw_single ( root, !dir );
        else if ( is_red ( root->link[dir]->link[!dir] ) )
          root = jsw_double ( root, !dir );
      }
    }
  }

  return root;
}

int jsw_insert ( struct jsw_tree *tree, int data )
{
  tree->root = jsw_insert_r ( tree->root, data );
  tree->root->red = 0;
  return 1;
}
```

Okay, that was anticlimactic. Simple, elegant, that's red black insertion from the bottom-up! I recommend you go through an example that covers all of the cases, just to make sure that it works. Yes, we have the test function, but how do you know I'm not just messing with you? Well, you'll know after you grab about 30 random numbers and do your own execution trace!

Insertion really is the easiest part of red black trees. A red violation is sickly simple to test for and fix without causing problems elsewhere. The big problems come when you bottom-up deletion. Get ready for a good time!

Bottom-up Deletion

Deletion of a node from a red black tree is a pain in the ass. That's my official opinion too. I hate it and you'll learn to hate it. Especially the bottom-up algorithm. Especially the first way I'm going to describe it to you. >:-)

During insertion we had the option of selecting the color of a new node to make our lives easier. We forced a red violation and fixed it. Red violations are easy to fix, and we took full advantage of that to produce a truly elegant recursive algorithm. When you want to delete a node, you don't have the option of choosing its color. If it's red, that's great! Red nodes can be removed without any violations. Why? Well, I can't think of a way to introduce a red violation by removing a red node from a valid tree, but you're welcome to look for one. Since a red node doesn't participate in the black height, removing a red node has no effect on the black height. Therefore, removing a red node cannot violate the rules of a red black tree.

If the node is black, that's a problem. Removing a black node is sure to cause a black violation just like inserting a black node, which is why we avoided that during insertion, and it could very likely cause a red violation too! Ouch. Well, let's remove the node first, then think about how to fix violations. The following is a simple recursive deletion from a red black tree. If the node to be deleted has less than two children, we just replace it with it's non-null child, or a null pointer if there are no children.

Conveniently enough, we might be able to get away without rebalancing at this point. If the node that we're going to delete is black and has one red child, we can color the child black and introduce no violations because we've reduced the case of removing a black node to the case of removing a red node, which is guaranteed to cause no violations. We'll call this case 0.

If the node has two children, we find its inorder predecessor and copy the predecessor's data into the node. Then we recursively delete the predecessor, since it's guaranteed to have at most one child. All in all, the algorithm is very pretty, even with the extra

framework added to facilitate rebalancing (the simple recoloring after deletion, the status flag, and the mysterious **jsw_remove_balance**). Notice that the root is colored black at the end only if the tree is not empty:

```
struct jsw_node *jsw_remove_r ( struct jsw_node *root, int data, int *done )
{
  if ( root == NULL )
    *done = 1;
  else {
    int dir;

    if ( root->data == data ) {
      if ( root->link[0] == NULL || root->link[1] == NULL ) {
        struct jsw_node *save =
          root->link[root->link[0] == NULL];

        /* Case 0 */
        if ( is_red ( root ) )
          *done = 1;
        else if ( is_red ( save ) ) {
          save->red = 0;
          *done = 1;
```



```
      }
      else {
        struct jsw_node *heir = root->link[0];

        while ( heir->link[1] != NULL )
          heir = heir->link[1];

        root->data = heir->data;
        data = heir->data;
      }
    }

    dir = root->data < data;
    root->link[dir] = jsw_remove_r ( root->link[dir], data, done );

    if ( !*done )
      root = jsw_remove_balance ( root, dir, done );
  }

  return root;
}

int jsw_remove ( struct jsw_tree *tree, int data )
{
  int done = 0;

  tree->root = jsw_remove_r ( tree->root, data, &done );
  if ( tree->root != NULL )
    tree->root->red = 0;

  return 1;
}
```

This time we really do need a status flag to tell the algorithm when to stop rebalancing. If **jsw_remove_balance** is called all of the way up the tree, it could get ugly. Since this function only removes external nodes (nodes without two children), we only have to worry about setting the flag after a removal (if the node is red or case 0 applies), and inside **jsw_remove_balance**.

It's obvious that the real work is done by **jsw_remove_balance**, but what does this helper function do? It handles all of the cases that could pop up after removing a black node, of course! And it's about as long as **jsw_remove_r**. :-(Let's look at the simple cases first. If we remove a node, and the node's sibling is black, and its children are both black, we have an easy case that propagates (* = the place we deleted):

```
            3,B                     3,B

          /     \                 /     \

      1,B          *  ->      1,R          *

    /     \                 /     \

 0,B         2,B        0,B         2,B
```

In this case, we can simply recolor 1 (the sibling) to be red without introducing a red violation, but the possibility still exists that there's a black violation because we decreased the black height of 3's left subtree. This causes both of 3's subtrees to be the

same black height, but if 3 itself is a subtree, then its parent will see a black violation since the sibling subtree hasn't decreased in black height. However, if the parent node (3

... the tree. We

```
          3,R                     3,B
        /     \                 /     \
    1,B         *  ->       1,R         *
   /   \                   /   \
0,B     2,B           0,B       2,B
```

Now for the harder cases where the sibling is black and either or both of its children are red. There are two cases here. If the left child of the sibling is red then we only need to perform a single rotation. However, the parent could be either red or black, so we need to save its color and restore it later, because a rotation doesn't consider the old colors of the new parent and old parent. After the rotation, the new parent is recolored with the old parent's color and its children become black (E = either color):

```
          3,E             1,E
        /     \         /     \
    1,B         *  -> 0,B       3,B
   /   \                      /     \
0,R     2,E               2,E         *
```

This restores the black height of the subtree, and the status flag can be set to signal that no more rebalancing is needed. The same process is followed if the sibling's right child is red, except this time we use a double rotation instead of a single rotation. The final colors are identical, and keep in mind that the sibling's left child would have to be black in this case. If it's red then the previous case applies:

```
          3,E                     2,E
        /     \                 /     \
    1,B         *  ->       1,B         3,B
   /   \                   /               \
0,B     2,R           0,B                   *
```

Whew, what a workout! But I have bad news for you. Those are only the cases where the sibling is black. If the sibling is red, it's even worse. But I have good news for you too. The following discussion doesn't matter except as an intellectual curiosity. We'll completely avoid the red sibling case shortly. So take a deep breath and get ready for the second half of rebalancing cases for deletion.

All set? Okay, every case for a red sibling requires at least one rotation, but no more than three. But first, the simple cases. If the sibling is red then both of its children are black, so we can do a single rotation, recolor the new parent black, and recolor the sibling's right child red to restore balance. Remember that the cases are symmetric, even though the descriptions and diagrams only consider deleting from the right subtree, but by using the direction index trick, we only need to handle one of the symmetric cases (where **dir** would be 1). All of this should become clear when you see the code:


```
   /   \                      /     \
0,B     2,B               2,R         *
```

It's obvious after seeing a diagram how this restores balance by fixing the black heights, but it's confusing why 2 becomes red. Well, the red has to go somewhere or the black height of the tree would be affected, and the only known node that we are sure can be made red is that particular child. I don't quite remember how I figured out the red sibling cases, but I'm reasonably sure that alcohol was involved. :-) The second red

sibling case looks at the inner child of the sibling. This child cannot be a null pointer (exercise: why?), so we simply test one of its children and act depending on which one is red. If the outer child is red, we perform a double rotation, color the new parent black, its right child black, and its left child red:

```
     3,B             2,B
   /    \           /    \
C,R       *  ->  0,R        3,B
   \                \          \
     2,B             1,B         *
   /
1,R
```

This restores the black height of the right subtree without changing the black height of the left subtree. When we have two red nodes to work with, it's easy to make one of them black after the rotation and leave the other to avoid violating the black height of the subtree that we took it from. The last case is if 3's right child is red. The good news is that we can reduce this to the previous case by a single rotation at 2, then a double rotation at 5 (the previous case) will give us the structure we want, with the same colors as above.

```
     3,B             3,B             2,B
   /    \           /    \          /    \
C,R       *  ->  0,R       *  ->  0,R       3,B
   \                \                \          \
     1,B             2,B              1,B         *
        \           /
          2,R    1,R
```

Before you start thinking of the terrible mess that **jsw_remove_balance** must be, let's look at it and see that it really isn't that bad. Yes, it's long, but none of the cases are overly difficult. The real savings come from noticing similarities between cases and setting colors outside of the cases so as to avoid repeating code, but we can do much better as you'll see shortly. Compare the cases described above with their translations into source code. Did we cover all of the cases?


```
if ( s != NULL && !is_red ( s ) ) {
  /* Black sibling cases */
  if ( !is_red ( s->link[0] ) && !is_red ( s->link[1] ) ) {
    if ( is_red ( p ) )
      *done = 1;
    p->red = 0;
    s->red = 1;
  }
  else {
    int save = root->red;

    if ( is_red ( s->link[!dir] ) )
      p = jsw_single ( p, dir );
    else
      p = jsw_double ( p, dir );

    p->red = save;
    p->link[0]->red = 0;
    p->link[1]->red = 0;
    *done = 1;
  }
}
else if ( s->link[dir] != NULL ) {
  /* Red sibling cases */
  struct jsw_node *r = s->link[dir];

  if ( !is_red ( r->link[0] ) && !is_red ( r->link[1] ) ) {
    p = jsw_single ( p, dir );
    p->link[dir]->link[!dir]->red = 1;
  }
  else {
```

```
      if ( is_red ( r->link[dir] ) )
        s->link[dir] = jsw_single ( r, !dir );
      p = jsw_double ( p, dir );
      s->link[dir]->red = 0;
      p->link[!dir]->red = 1;
    }

    p->red = 0;
    p->link[dir]->red = 0;
    *done = 1;
  }

  return p;
}
```

Traditionally, red black trees reduce the red sibling case to a black sibling case. Unless you're familiar with the red sibling case (which is why I showed it to you), the reasons behind this case reduction would be terribly confusing. But, because the sibling's parent and children have to be black, a single rotation will push the parent down on the side that we deleted, recolor it red, pull the sibling and its children up, and recolor the sibling black. This doesn't change the black height of the tree, it just reverses which side the violation is on. By pushing the entire violation down, we ensure that the new sibling (2 in this case) is black:

```
         3,B              1,B

       /     \          /     \

    1,R         *  ->  0,B        3,R

  /     \                       /     \
```


need to move down along with the sibling. So * would still be the place we removed a node, but 2 would be the new sibling. The cases after this are identical, but this time we need to be careful to move down without losing anything so that we can move back up without any trouble:

```
struct jsw_node *jsw_remove_balance ( struct jsw_node *root, int dir, int *done )
{
  struct jsw_node *p = root;
  struct jsw_node *s = root->link[!dir];

  /* Case reduction, remove red sibling */
  if ( is_red ( s ) ) {
    root = jsw_single ( root, dir );
    s = p->link[!dir];
  }

  if ( s != NULL ) {
    if ( !is_red ( s->link[0] ) && !is_red ( s->link[1] ) ) {
      if ( is_red ( p ) )
        *done = 1;
      p->red = 0;
      s->red = 1;
    }
    else {
      int save = p->red;
      int new_root = ( root == p );

      if ( is_red ( s->link[!dir] ) )
        p = jsw_single ( p, dir );
      else
        p = jsw_double ( p, dir );

      p->red = save;
      p->link[0]->red = 0;
      p->link[1]->red = 0;

      if ( new_root )
        root = p;
      else
        root->link[dir] = p;

      *done = 1;
    }
  }

  return root;
}
```

The subtle waltz of **root** and **p** could be confusing, but the only fix for that is an execution trace to make sure that you understand why **root** is used where it is and why **p** is used where it is. Remember that I said the case reduction was confusing (probably why nobody cares to explain it), but it simplifies the code greatly by removing half of the rebalancing work that we need to do.

Top-down Insertion

One of the primary benefits of red black trees is that both the insertion and deletion algorithms can be written in one top-down pass, instead of an algorithm that walks

useful work on ... changes and

This means that we don't need to use recursion any longer, or any other tricks that make the algorithms confusing. So let's look at a non-recursive top-down insertion.

One key concept that you need to take away from this tutorial is that the rules are the same, regardless of how you implement them. So there are still three cases for red black insertion: color flip, single rotation, and double rotation. Everything else is framework and details. However, for top-down insertion we don't have the benefit of riding the recursion wave. To recover from that deficiency we need to carefully save several pointers back up the tree. At the very least we have the current node along the path (which will be a null pointer at the end), its parent, and the parent's parent.

To avoid special cases, we'll also use a dummy tree root, where the actual root is the dummy's right link. This way we don't have to constantly test the special boundary case of rebalancing at the root. To make rotations simpler, we'll also add a pointer for the parent of the grandparent. So at the most we're saving the last three levels of the path.

For the color flip case, we test the current node's children. If they're red then the current node must be black, so we do a color flip. This could cause a red violation further up the tree (which is why we saved the parent of the current node). Now, because a color flip could immediately cause a violation, we go straight to the violation test between the parent and the current node. If both are red, we do a single or double rotation at the grandparent:

```
Color Flip

1,B <-- p                 1,B

   \                         \

     3,B <-- q  becomes        3,R

   /     \                   /     \

2,R         4,R           2,B         4,B


Single Rotation

0,B <-- g                           1,B

   \                              /     \

     1,R <-- p                0,R          3,R

         \            becomes           /     \

           3,R <-- q                2,B          4,B

         /     \

     2,B          4,B

Double Rotation

     0,B <-- g                          2,B

         \                           /           \
```

```
  4,R <-- p          0,R          4,R
```



```
     /        \
   1,B        3,B
```

Notice how a rotation is made around the grandparent, which is why we also save a pointer to the grandparent's parent. Otherwise it would be rather difficult to notify the tree of the changes we're making. The code to perform this magic is shorter than you would expect, but it doesn't follow conventional methods. All of the work is done in the walk down loop, including insertion, since inserting a new red node could cause a violation. So the loop breaks when the data is found either because a new node was inserted or because there was already a node with that data in the tree. As such, the following algorithm does not allow duplicates and also does not warn about a duplicate:

```
int jsw_insert ( struct jsw_tree *tree, int data )
{
  if ( tree->root == NULL ) {
    /* Empty tree case */
    tree->root = make_node ( data );
    if ( tree->root == NULL )
      return 0;
  }
  else {
    struct jsw_node head = {0}; /* False tree root */

    struct jsw_node *g, *t;     /* Grandparent & parent */
    struct jsw_node *p, *q;     /* Iterator & parent */
    int dir = 0, last;

    /* Set up helpers */
    t = &head;
    g = p = NULL;
    q = t->link[1] = tree->root;

    /* Search down the tree */
    for ( ; ; ) {
      if ( q == NULL ) {
        /* Insert new node at the bottom */
        p->link[dir] = q = make_node ( data );
        if ( q == NULL )
          return 0;
      }
      else if ( is_red ( q->link[0] ) && is_red ( q->link[1] ) ) {
        /* Color flip */
        q->red = 1;
        q->link[0]->red = 0;
        q->link[1]->red = 0;
      }

      /* Fix red violation */
      if ( is_red ( q ) && is_red ( p ) ) {
        int dir2 = t->link[1] == g;

        if ( q == p->link[last] )
          t->link[dir2] = jsw_single ( g, !last );
        else
          t->link[dir2] = jsw_double ( g, !last );
      }

      /* Stop if found */
      if ( q->data == data )
        break;

      last = dir;
```

http://www.eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx Go SEP NOV JAN
125 captures 12 Feb 2007 - 8 Feb 2021 2013 29 2014 2015 About this capture

```
      g = p, p = q;
      q = q->link[dir];
    }

    /* Update root */
    tree->root = head.link[1];
  }

  /* Make root black */
  tree->root->red = 0;

  return 1;
}
```

Granted, this top-down insertion isn't as pretty as the recursive bottom-up insertion, but it takes full advantage of the properties of red black trees, and avoids the potential problems of recursion. It's also theoretically more efficient. :-) The good news is that deletion is easier top-down than bottom-up, and we can use the lessons gained from insertion to make implementing it easier. Let's take a look.

Top-down Deletion

Bottom-up deletion was a pain because we needed to recover from a black violation, which is the harder of the two violations to fix. However, the deletion was trivial if we removed a red node because deleting a red node can't violate any of the rules of a red black tree. If we could somehow guarantee that the node to be deleted was red, it would simplify deletion greatly. The good news is that we can do just that! By forcing a red node into the tree at the top and pushing it down using color flips and rotations, we can ensure that a red node will be deleted, always.

There are only four cases for pushing a red node down the tree without a black violation. The first case is a simple reverse color flip. If a node and it's sibling are black, and all four of their children are black, make the parent black and both children red:

```
    1,R             1,B

  /     \     ->  /     \

0,B       2,B   0,R       2,R
```

This is basically a reverse color flip from what we did during insertion. It pushes the red nodes down rather than forcing them up, all without changing the black height of the tree. Cool. :-) The next case is the dreaded red sibling case, which we will quickly reduce to nothing with a single rotation, using the tricks we learned from bottom-up deletion. This case is very intuitive now that we don't have to change directions to get it to work. Notice how the black heights don't change, nor does the color of the parent. Since we're moving down already, pushing the sibling down is just what the doctor ordered:

```
    1,B         0,B

  /     \          \

0,R       2,B ->    1,R
```


The final two cases are, you guessed it, single and double rotations depending on the color of the sibling's children. The cases are (should be!) familiar by now, so we'll just go over them briefly before looking at the code. If the right child is red, we perform a double rotation. However, in this case the color changes of **jsw_single** won't quite cut it. To be thorough, we'll force the correct coloring for all affected nodes:

```
    2,R             1,R

  /     \         /     \

0,B       3,B -> 0,B      2,B

  \                         \

    1,R                       3,R
```

As always, we have to make sure that the color of the parent doesn't change and the black heights of both subtrees remain the same as they were originally. These two guidelines avoid both red and black violations. If the left child of the sibling is red then a single rotation is more than enough to do the same thing. Since the final colors are the same, so the recoloring of this case can be lumped together with the previous case:

```
        2,R             1,R

      /     \         /     \

    1,B       3,B -> 0,B      2,B

  /                             \

0,R                               3,R
```

Now for the fun part. This algorithm will use a sneaky deletion by copying technique that saves the node to be deleted, and just keeps on going through the loop until the inorder predecessor is found. At that point the walk down loop terminates, the data items are copied and the external node is spliced out of the tree. This is as opposed to a naive deletion by copying that breaks the search off when the node is found, then has an extra step that explicitly looks for the predecessor. On the way down, of course, we look for and handle the cases shown above:

```
int jsw_remove ( struct jsw_tree *tree, int data )
{
  if ( tree->root != NULL ) {
    struct jsw_node head = {0}; /* False tree root */
    struct jsw_node *q, *p, *g; /* Helpers */
    struct jsw_node *f = NULL;  /* Found item */
    int dir = 1;

    /* Set up helpers */
    q = &head;
    g = p = NULL;
    q->link[1] = tree->root;

    /* Search and push a red down */
    while ( q->link[dir] != NULL ) {
      int last = dir;

      /* Update helpers */
      g = p, p = q;
      q = q->link[dir];
```



```
      /* Push the red node down */
      if ( !is_red ( q ) && !is_red ( q->link[dir] ) ) {
        if ( is_red ( q->link[!dir] ) )
          p = p->link[last] = jsw_single ( q, dir );
        else if ( !is_red ( q->link[!dir] ) ) {
          struct jsw_node *s = p->link[!last];

          if ( s != NULL ) {
            if ( !is_red ( s->link[!last] ) && !is_red ( s->link[last] ) ) {
              /* Color flip */
              p->red = 0;
              s->red = 1;
              q->red = 1;
            }
            else {
              int dir2 = g->link[1] == p;

              if ( is_red ( s->link[last] ) )
                g->link[dir2] = jsw_double ( p, last );
              else if ( is_red ( s->link[!last] ) )
                g->link[dir2] = jsw_single ( p, last );

              /* Ensure correct coloring */
              q->red = g->link[dir2]->red = 1;
              g->link[dir2]->link[0]->red = 0;
              g->link[dir2]->link[1]->red = 0;
            }
          }
        }
      }
    }

    /* Replace and remove if found */
    if ( f != NULL ) {
      f->data = q->data;
      p->link[p->link[1] == q] =
        q->link[q->link[0] == NULL];
      free ( q );
    }

```



```
    if ( tree->root != NULL )

  return 1;
}
```

It's short and sweet, though the extreme level of indention is a little distasteful for some. As an exercise, try to shrink that down. :-) As with insertion we used a dummy root to avoid special cases, and a few helper variables to point up the tree. The majority of the complexity comes from our rebalancing cases, and if you remove them, you'll

discover a very clean deletion algorithm for a basic binary search tree. Be sure to trace the execution of this function to make sure that you understand how it works, because it's a very nifty algorithm. Though I'm a little biased because I've never seen it used elsewhere, which means I may be the first to discover it. :-)

That's top-down deletion, and it can be compared to bottom-up deletion in the same ways as insertion, but now we can also consider that the top-down code is shorter and (I think) simpler. Red black trees are generally cleaner and easier to understand if written in a top-down manner, but unfortunately, due to the fact that the only well known resource that gives code for deletion is CLR, the most common implementations are still bottom-up. Oh well.

Conclusion

Red black trees are interesting beasts. They're believed to be simpler than AVL trees (their direct competitor), and at first glance this seems to be the case because insertion is a breeze. However, when one begins to play with the deletion algorithm, red black trees become very tricky. However, the counterweight to this added complexity is that both insertion and deletion can be implemented using a single pass, top-down algorithm. Such is not the case with AVL trees, where only the insertion algorithm can be written top-down. Deletion from an AVL tree requires a bottom-up algorithm.

So when do you use a red black tree? That's really your decision, but I've found that red black trees are best suited to largely random data that has occasional degenerate runs, and searches have no locality of reference. This takes full advantage of the minimal work that red black trees perform to maintain balance compared to AVL trees and still allows for speedy searches.

Red black trees are popular, as most data structures with a whimsical name. For example, in Java and C++, the library map structures are typically implemented with a red black tree. Red black trees are also comparable in speed to AVL trees. While the balance is not quite as good, the work it takes to maintain balance is usually better in a red black tree. There are a few misconceptions floating around, but for the most part the hype about red black trees is accurate.

This tutorial described the red black abstraction for balancing binary search trees. Both insertion and deletion were covered in their top-down and bottom-up forms, and a full implementation for each form was given. For further information on red black trees, feel free to surf over to Ben Pfaff's excellent online book about libavl (I wish I had it when figuring this stuff out!), or pick up "Introduction to Algorithms" by Cormen, Leiserson, and Rivest.

From the twisted mind of Julienne Walker

Top